Indexing Techniques for First-Order Theorem Proving

Stephan Schulz
schulz@eprover.org

Idea

Quickly find inference partners in large search states

- Replace linear search with index access
- Especially valuable for simplifying inferences

More concretely (or more abstractly?):

- Given a set of terms or clauses S
- and a query term or query clause
- and a retrieval relation R
- Build a data structure to efficiently find (all) terms or clauses t from S such that $R(t, S)$ (the retrieval relation holds)

Background

Setting:

- First order clausal logic (with equality)
- Saturation based theorem proving
- Proof procedure enumerates logical consequences looking for a contradiction

Data types:

- First-order terms, e.g. a, $f(X, g(Y))$, $g(g(g(a)))$
- Equational atoms: $t_1 \simeq t_2$ (where the t_i are terms)
- (Plain atoms: $p(t_1, \ldots, t_n) \equiv p(t_1, \ldots, t_n) \simeq \top$)
- Literals: Atoms or negated atoms $t_1 \not\simeq t_2$, $\neg p(t_1)$,
- Clauses:
  - Formally: (Multi-)sets of literals: $\{p(a), f(X) \simeq Y, X \not\simeq Y\}$
  - Written: $p(a) \vee f(X) \simeq Y \vee X \not\simeq Y$
  - Interpreted as disjunctions
- Notice: $\simeq, \not\simeq$ are commutative, $\vee$ is AC

Search state is a set of clauses (implicitely conjunctively connected)

Introductory Example: Text Indexing

Problem: Given a set `D` of text documents, find all documents that contain a certain word `w`

Obviously correct implementation:

```
result = {}
for doc in D
      for word in doc
            if w == word
                  result = result ∪{ doc }
                  break;
return result
```

Now think of Google. . .

- Obvious approach (linear scan through documents) breaks down for large D
- Instead: Precompiled Index $I : words \rightarrow documents$
- Requirement: I efficiently computable for large number of words!

The Trie Data Structure

Definition: Let Σ be a finite alphabet and Σ^* the set of all words over Σ

- We write $|w|$ for the length of w
- If $u, v \in \Sigma^*$, $w = uv$ is the word with prefix u

A trie is a finite tree whose edges are labelled with letters from Σ

- A node represents a set of words with a common prefix (defined by the labels on the path from the root to the node)
- A leaf represents a single word
- The whole trie represents the set of words at its leaves
- Dually, for each set of words S (such that no word is the prefix of another), there is a unique trie T

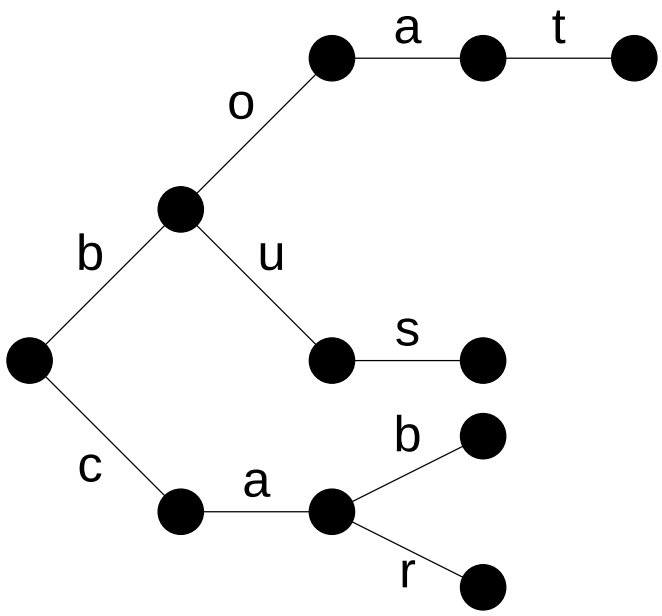
Fact: Finding the leaf representing w in T (if any) can be done in $O(|w|)$

- This is independent of the size of S!
- Inserting and deleting of elements is just as fast

Trie Example

Consider $\Sigma = \{a, b, ..., z\}$ and $S = \{car, cab, bus, boat\}$

The trie for S is:



Tries can be built incrementally

We can store extra infomation at nodes/leaves

- E.g. all documents in which $boat$ occurs
- Retrieving this information is fast and simple

Indexing Techniques for Theorem Provers

Term Indexing standard technique for high performance theorem provers

- Preprocess term sets into index
- Return terms in a certain relation to a query term
  - Matches query term (find generalizations)
  - Matched by query term (find specializations)

Perfect indexing:

- Returns exactly the desired set of terms
- May even return substitution

Non-perfect indexing:

- Returns candidates (superset of desired terms)
- Separate test if candiate is solution

Frequent Operations

Let S be a set of clauses

Given term t, find an applicable rewrite rule in S

- Forward rewriting
- Reduced to: Given t, find $l \simeq r \in S$ such that $l\sigma = t$ for some σ
- Find generalizations

Given $l \rightarrow r$, find all rewritable clauses in S

- Backward rewriting
- Reduced to: Given l, find t such that $C|_p\sigma = l$
- Find instances

Given C, find a subsuming clause in S

- Forward subsumption
- Not easily reduced. . .
- Backward subsumption analoguous

Classification of Indexing Techniques

Perfect indexing

- The index returns exactly the elements that fullfil the retrieval condition
- Examples:
  - Perfect discrimination trees
  - Substitution trees
  - Context trees

Non-perfect indexing:

- The index returns a superset of the elements that fullfil the retrieval condition
- Retrieval condition has to be verified
- Examples:
  - (Non-perfect) discrimination trees
  - (Non-perfect) Path indexing
  - Top-symbol hashing
  - Feature vector-indexing

The Given Clause Algorithm

U: Unprocessed (passive) clauses (initially Specification)
P: Processed (active) clauses (initially: empty)

```
while U ≠ {}
    g = delete_best(U)
    g = simplify(g,P)
    if g == □
        SUCCESS, Proof found
    if g is not redundant w.r.t. P
        P = P\{c ∈ P|c redundant w.r.t. g}
        T = {c ∈ P|c simplifiable with g}
        P = (P\T) ∪ {g}
        T = T∪generate(g, P)
        foreach c ∈ T
            c =simplify(c, P)
            if c is not trivial
                U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Typically, $|U| \sim |P|^2$ and $|U| \approx \sum |T|$

The Given Clause Algorithm

U: Unprocessed (passive) clauses (initially Specification)
P: Processed (active) clauses (initially: empty)

```
while U ≠ {}
    g = delete_best(U)
    g = simplify(g,P)
    if g == □
        SUCCESS, Proof found
    if g is not redundant w.r.t. P
        P = P\{c ∈ P|c redundant w.r.t. g}
        T = {c ∈ P|c simplifiable with g}
        P = (P\T) ∪ {g}
        T = T∪generate(g, P)
        foreach c ∈ T
            c =simplify(c, P)
            if c is not trivial
                U = U ∪ {c}
SUCCESS, original U is satisfiable
```

Simplification of new clauses is bottleneck

Sequential Search for Forward Rewriting

Given t, find $l \simeq r \in S$ such that $l\sigma = t$ for some σ

Naive implementation (e.g. DISCOUNT):

function find_matching_rule(t, S)
    for $l \simeq r \in S$
        σ = match(l, t)
        if σ and $l\sigma > r\sigma$
            return (σ, $l \simeq r$)

Remark: We assume that for unorientable $l \simeq r$, both $l \simeq r$ and $r \simeq l$ are in S

Conventional Matching

match(s,t)
    return match_list($[s], [t], \{\}$)

match_list(ls, lt, σ)
    while $ls \neq []$
        $s =$ head(ls)
        $t =$ head(lt)
        if $s == X \in V$
            if $X \leftarrow t' \in \sigma$
                if $t \neq t'$ return FAIL
                else
                    $\sigma = \sigma \cup \{X \leftarrow t\}$
        else if $t == X \in V$ return FAIL
        else
            let $s = f(s_1, \ldots, s_n)$
            let $t = g(t_1, \ldots, t_m)$
            if $f \neq g$ return FAIL /* Otherwise $n = m$! */
        $ls =$ append(tail(ls), $[s_1, \ldots s_n]$
        $lt =$ append(tail(lt), $[t_1, \ldots t_m]$)
    return σ

The Size of the Problem

Example LUSK6:

- Run time with E on 1GHz Powerbook: 1.7 seconds
- Final size of P: 265 clauses (processed: 1542)
- Final size of U: 26154 clauses
- Approximately 150,000 successful rewrite steps
- Naive implementation: $\approx$ 50-150 times more match attempts!
- $\approx$ 100 machine instructions/match attempt

Hard examples:

- Several hours on 3+GHz machines
- Billions of rewrite attempts

Naive implementations don't cut it!

Top Symbol Hashing

Simple, non-perfect indexing method for (forward-) rewriting

Idea: If $t = f(t_1, \ldots, t_n)$ ($n \geq 0$), then any s that matches t has to start with f

– $top(t) = f$ is called the top symbol of t

Implementation:

– Organize $S = \cup S_f$ with $S_f = \{l \simeq r \in S | top(l) = f\}$
– For non-variable query term t, test only rewrite rules from $S_{top(t)}$

Efficiency depends on problem composition

– Few function symbols: Little improvement
– Large signatures: Huge gain
– Typically: Speed-up factor 5-15 for matching

String Terms and Flat Terms

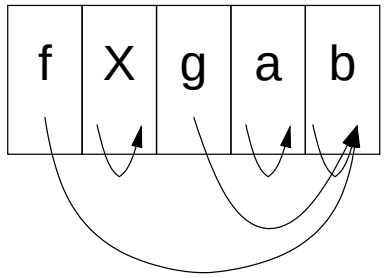
Terms are (conceptually) ordered trees

- Recursive data structure
- But: Conventional matching always does left-right traversal
- Many other operations do likewise

Alternative representation: String terms

- $f(X, g(a, b))$ already is a string. . .
- If arity of function symbols is fixed, we can drop braces: f$Xgab$
- Left-right iteration is much faster (and simpler) for string terms

Flat terms: Like string terms, but with term end pointers



- Allows fast jumping over subterms for matching

Perfect discrimination tree indexing

Generalization of top symbol hashing

Idea: Share common prefixes of terms in string representation

- Represent terms as strings
- Store string terms (left hand sides of rules) in trie (perfect discrimination tree)
- Recursively traverse trie to find matching terms for a query:
  - At each node, follow all compatible vertices in turn
  - If following a variable branch, add binding for variable
  - If no valid possibility, backtrack to last open choice point
  - If leaf is reached, report match

Currently most frequently used indexing technique

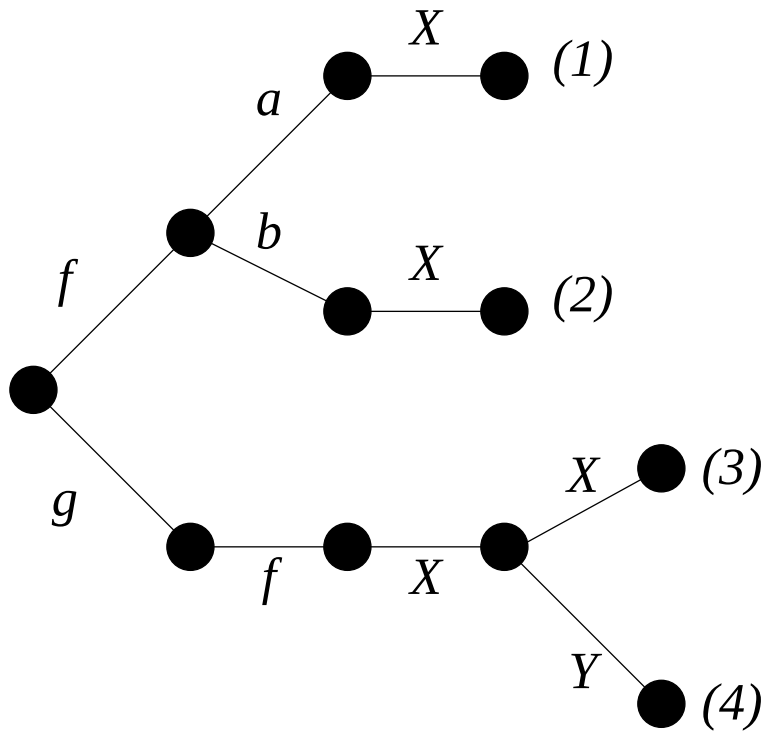
- E (rewriting, unit subsumption)
- Vampire (rewriting, unit- and non-unit subsumption (as code trees))
- Waldmeister (rewriting, unit subsumption, paramodulation)
- Gandalf (rewriting, subsumption)
- ...

Example

Consider $S = \{(1) f(a, X) \simeq a, (2) f(b, X) \simeq X,$
$(3) g(f(X, X)) \simeq f(Y, X), (4) g(f(X, Y)) \simeq g(X)\}$

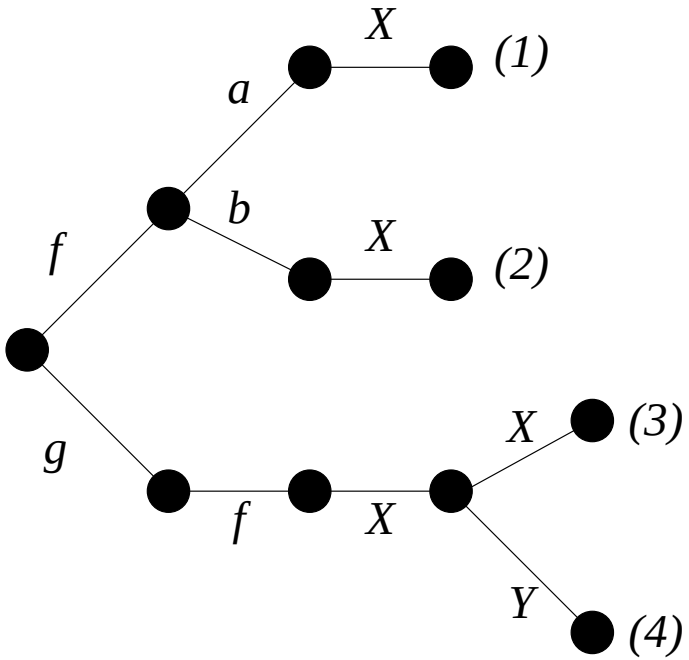
– String representation of left hand sides: $faX, fbX, gfXX, gfXY$

– Corresponding trie:



Find matching rule for $g(f(a, g(b)))$

Example Continued



Start with $g(f(a, g(b)))$, root node, $\sigma = \{\}$

$g(f(a, g(b)))$ Follow g vertex
$g(f(a, g(b)))$ Follow f vertex
$g(f(a, g(b)))$ Follow X vertex, $\sigma = \{X \leftarrow a\}$, jump over a
$g(f(a, g(b)))$

- Follow X vertex - Conflict! X already bound to a
- Follow Y, $\sigma = \{X \leftarrow a, Y \leftarrow g(b)\}$, jump over $g(b)$ Rule 4 matches

Subsumption Indexing

Subsumption: Important simplification technique for first-order reasoning

- Drop less general (redundant) clauses
- Keep more general clause

Problem: Efficiently detecting subsumed clauses

- Individual clause-clause subsumption is in NP
- Large number of subsumption relations must be tested

Major Approach: Indexing

- Use precompiled data structures to efficiently select
  - subsuming clauses (forward subsumption)
  - subsumes clause (backward subsumption)

  from large (and fairly static) clause sets

Usual: Different and complex indexing approaches for forward- and backward subsumption

Subsumption

Idea: Only keep the most general clauses

– If one clause is subsumed by another, discard it

Formally: A clause C subsumes C' if:

– There exists a substitution σ such that $C\sigma \subseteq C'$
– Note: In that case $C \models C'$
– $\subseteq$ usually is the multi-subset relation

Examples:

– $p(X)$ subsumes $p(a) \vee q(f(X), a)$ ($\sigma = \{X \leftarrow a\}$)
– $p(X) \vee p(Y)$ does not multi-set-subsume $p(a) \vee q(f(X), a)$
– $q(X, Y) \vee q(X, a)$ subsumes $q(a, a) \vee q(a, b)$

Subsumption is hard (NP-complete)

– $n!$ permutations in non-equational clause with n literals
– $n!2^n$ permutations in equational clause with n literals

Forward- and Backward Subsumption

Assume a set of clauses P and a given clause p

Forward subsumption: Is there any clause in P that subsumes g?

Backward subsumption: Find/remove all clauses in P subsumed by g

Notice that these are clause–clause set operations

Naive implementation: Sequence of clause-clause operations

– Good implementation can speed up (average case) individual subsumption
– Number of attempts still very high

Smarter: Avoid many of the subsumption calls up front

– Use indexing techniques to reduce number of candidates

Feature Vector Indexing

New clause indexing technique

– Not lifted from term indexing

Advantages:

– Small index (memory footprint)
– Same index for forward- and backward subsumption
– Very simple
– Efficient in practice
– Variants for different subsumption relations

Disadvantages:

– Non-perfect
– Requires fixed signature for optimal performance

How does it work?

Properties of the Subsumption Relation

Definitions:

- Let C and C' be clauses
- C^+ is the (multi-)set (a clause) of positive literals in C
- C^- is the (multi-)set of negative literals in C
- $|C|_f$ is the number of occurences of (function or predicate) symbol f in C

Facts: If C subsumes C', then

- $|C^+| \leq |C'^+|$
- $|C^-| \leq |C'^-|$
- $|C^+|_f \leq |C'^+|_f$ for all f
- $|C^-|_f \leq |C'^-|_f$ for all f
- (Similar results exist for term depths)
- The same holds for all linear combination of these features

Remark: Composite critera are often used to detect subsumption failure early

- $|C| \leq |C'|$ (C cannot have more literals than C')
- $\sum_{f \in F} |C|_f \leq \sum_{f \in F} |C'|$ (C cannot have more symbols than C')

Feature Vectors

Definitions:

- A feature function f is a function from the set of clauses to $\mathbf{N}$
- f is subsumption-compatible, if C subsumes C' implies $f(C) \leq f(C')$
- A (subsumption-compatible) feature vector function F is a function from the set of clauses to $\mathbf{N}^n$ such that $\Pi_n^i \circ F$ (the projection of F to the ith component) is a subsumption-compatible feature function
- If v_1 and v_2 are feature vectors, we write $v_1 \leq_s v_2$, if $v_1[i] \leq v_2[i]$ for all i.

Fact:

- Assume F is a (subsumption-compatible) feature vector function
- Assume C subsumes C'
- By construction, $F(C) \leq_s F(C')$

Basic Principle of Feature Vector Indexing:

- For forward-subsumption: $candFS_F(P, g) = \{c \in P | F(c) \leq_s F(g)\}$
- For backward-subsumption: $candBS_F(P, g) = \{c \in P | F(g) \leq_s F(c)\}$

Feature Vector Indexing

Aim: Efficiently compute $candF_F(P, g)$ and $candBS_F(P, g)$

Solution: Frequency vectors for P are compiled into a trie, clauses are stored in leaves

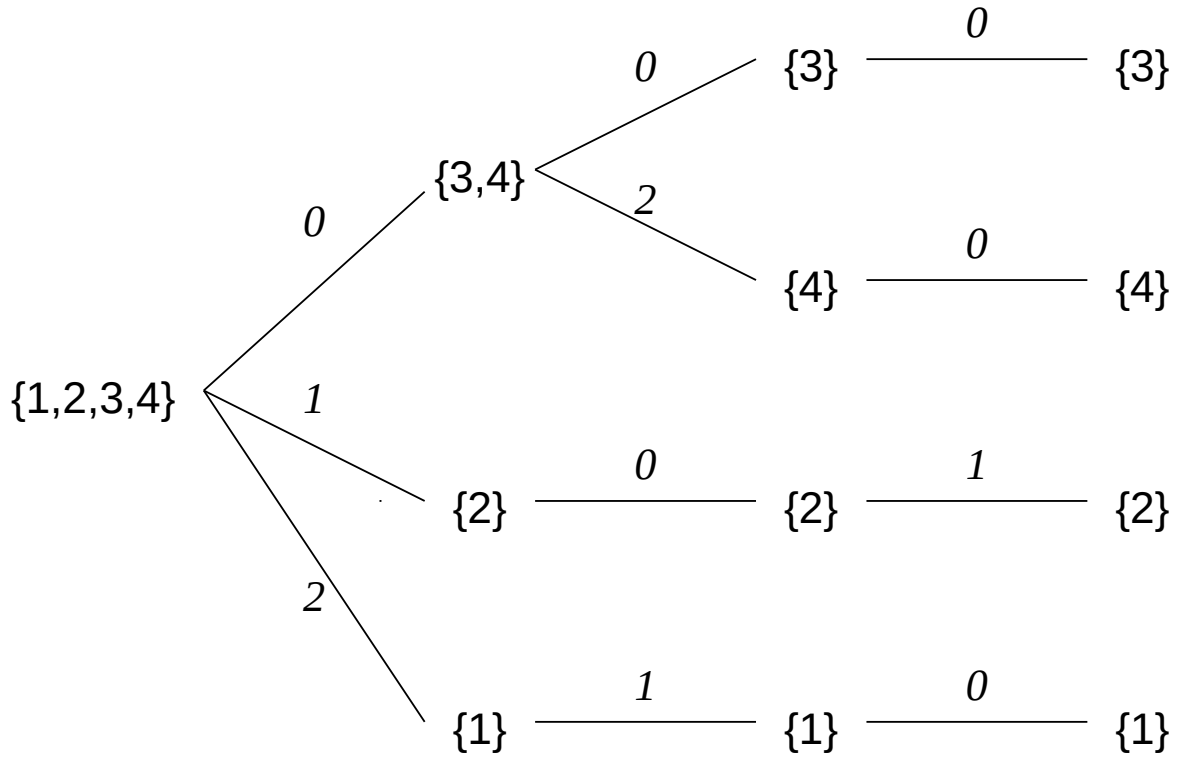
- Tree of depth n (number of features in vector)
- Nodes at depth d split according to feature $F(C)[d]$ (one successor per value)
- All vectors with value $F(C)[d] = k$ associated with corresponding subtree
- Construction continues recursively

Example: Assume $F(C) := \langle |C^+|_a, |C^+|_f, |C^-|_b| \rangle$

- Clause set $P = \{1,2,3,4\}$ with
  1. $F(p(a) \vee p(f(a))) = \langle 2, 1, 0 \rangle$
  2. $F(p(a) \vee \neg p(b)) = \langle 1, 0, 1 \rangle$
  3. $F(\neg p(a) \vee p(b)) = \langle 0, 0, 0 \rangle$
  4. $F(p(X) \vee p(f(f(b)))) = \langle 0, 2, 0 \rangle$
- Query $g = p(f(a))$
  - $F(g) = \langle 1, 1, 0 \rangle$

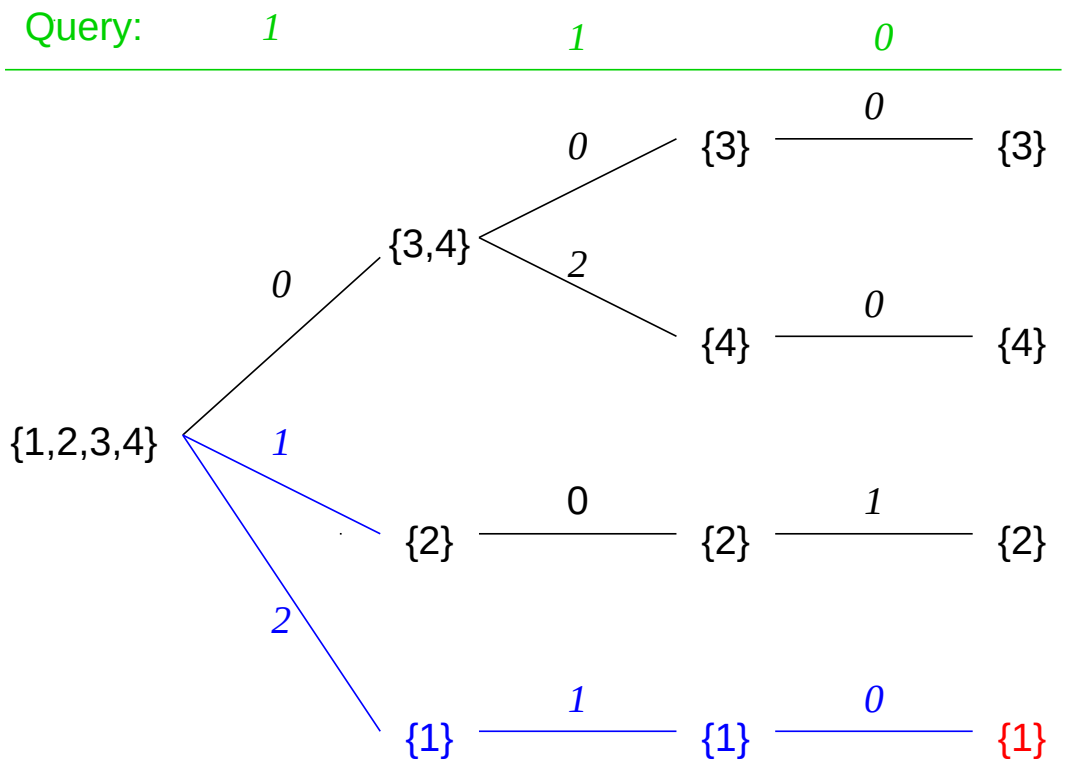
Example Index

1. $F(p(a) \vee p(f(a))) = \langle 2, 1, 0 \rangle$
2. $F(p(a) \vee \neg p(b)) = \langle 1, 0, 1 \rangle$
3. $F(\neg p(a) \vee p(b)) = \langle 0, 0, 0 \rangle$
4. $F(p(X) \vee p(f(f(b)))) = \langle 0, 2, 0 \rangle$



Example: Backward Subsumption

Algorithm: At each node, only follow branches with larger or equal feature values

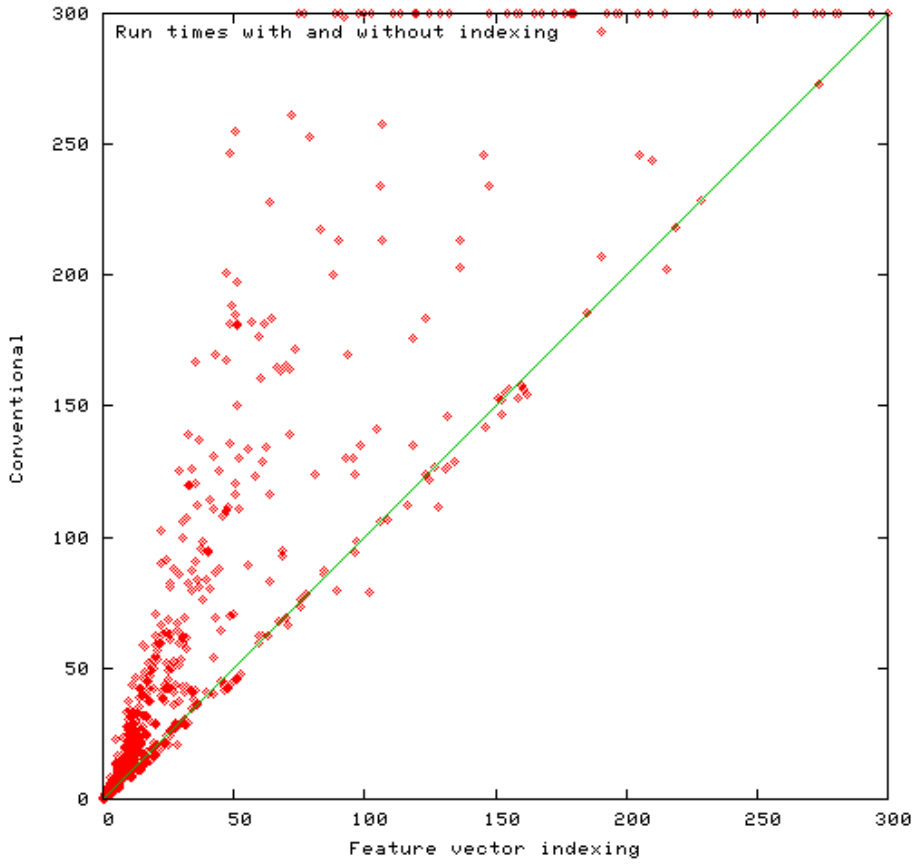


Result: Just one subsumption candidate for $p(f(a))$

Performance 1

Tested on 5180 examples from TPTP 2.5.1

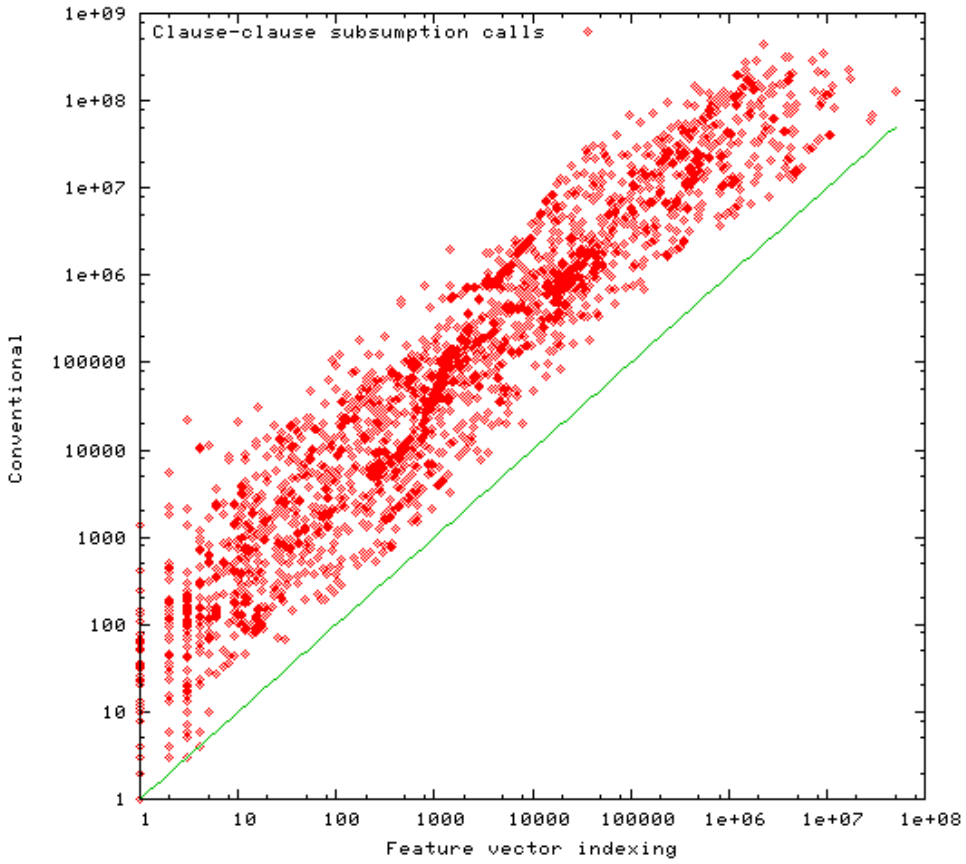
- Subsumption-heavy search strategy (contextual literal cutting)
- Max. 75 features, 300MHz SUN Ultra 60, 300s time limit



Speedup ca. 40%, overhead usually insignificant, 2717 vs. 2671 solutions found

Performance 2

Number of subsumption attempts (notice double log scale)



Average reduction: $1:60$, max: $1:8000 (1:\infty)$

Further Reading

References

[1] Graf, P., "Term Indexing," LNAI **1053**, Springer, 1995.

[2] Graf, P. and D. Fehrer, *Term Indexing*, in: W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, Applied Logic Series **9 (2)**, Kluwer Academic Publishers, 1998 pp. 125–147.

[3] Sekar, R., I. Ramakrishnan and A. Voronkov, *Term Indexing*, in: A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, II, Elsevier Science and MIT Press, 2001 pp. 1853–1961.

[4] Mark E. Stickel. *The Path-Indexing Method for Indexing Terms.* Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, USA, October 1989.

[5] McCune, W., *Experiments with Discrimination-Tree Indexing and Path In-*

dexing for Term Retrieval, Journal of Automated Reasoning **9** (1992), pp. 147–167.

[6] J. Christian. *Flat Terms, Discrimination Nets and Fast Term Rewriting. Journal of Automated Reasoning*, 10(1):95–113, 1993.

[7] Nieuwenhuis, R., T. Hillenbrand, A. Riazanov and A. Voronkov, *On the Evaluation of Indexing Techniques for Theorem Proving*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, LNAI **2083** (2001), pp. 257–271.

[8] Riazanov, A. and A. Voronkov, *Efficient Instance Retrieval With Standard and Relational Path Indexing*, in: F. Bader, editor, *Proc. of the 19th CADE, Miami*, LNAI **2741** (2003), pp. 380–396.

[9] Tammet, T., *Towards Efficient Subsumption*, in: C. Kirchner and H. Kirchner, editors, *Proc. of the 15th CADE, Lindau*, LNAI **1421** (1998), pp. 427–441.

[10] Schulz, S.: *Simple and Efficient Clause Subsumption with Feature Vector Indexing*, in: G. Sutcliffe and S. Schulz and T. Tammet, editor, *Proc. of*

the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland, Elsevier Science, ENTCS, 2004